The Token Test

Andrew Winterman

February 25, 2011

The Problem

- There is a plain text input file on a *nix system.
- Each line in the file has a token (string of arbitrary chars).
- The input file is very, very, very large in terms of number of lines.
- Your program has as much RAM as needed.
- Your program may make only one pass over the data, reading each line just once.
- At the end of the first pass, the program prints a report for each unique token it found. It prints out the token value itself and the number of times the token was found in the data file.
- The sort order of the output does not matter, but each unique token may appear only once in the output.
- Scalability is important.

Questions to Address:

- What algorithmic approach would you take and what data structures would you use?
- Why is it the best (fastest) solution?

General Idea

The general idea is to construct a linked list binary search tree which will hold information about the tokens discovered. First, a one-to-one function $g : \{\text{Tokens}\} \to \mathbb{N}$ is defined to generate keys based on tokens found, organizing the binary search tree. The binary search tree is a linked list in which each record is an array with five entries: address, token name, a counter of how many times the token has been found, and of course, pointers to the address of each of its children. The last entries of the array - the pointers to its children - are possibly empty. Every other element must have a value. Upon finding a

token the algorithm hunts for the a key matching the token, if it finds the token in the binary search tree, it increments the counter. If it fails to find the token, the algorithm inserts a node for the token where it expected to find it. I use the binary search tree because both search and insert take log m (where m is the length of the tree) time.

Run Time Analysis

To avoid page breaks in awkward places, the algorithm is presented on the next page, after this analysis of its performance. Suppose there are n lines in the program, and it discovers m tokens. Then the first part of the while loop takes $n *$ (time to read a line). Each time a token is discovered, it takes, on average $O(\log k)$, where k is the number of tokens discovered so far, to search, and O(1) time to insert a node or increment a counter. Hence, discovering all m tokens takes $O(\log\, m!)$. Reporting the token names and counter values takes $O(m)$ time, because each node of the tree must be visited. Hence the time complexity of the program is $O(n \cdot m\, \log\, m + m) = O(n\, m \log m)$.

Algorithm

```
WHILE(File.txt not ended)
   S <- checkForToken(current line)
```
Checks the current line for tokens, if a is token found, returns the token, else empty string
```
   IF( S == Empty String)
       Increment line
   ELSE
       M <- searchTree(B, S)
```
Conduct a binary search[1] of the binary search tree, B, for a token matching S. The function returns: `(TRUE, S.address, S.counter)` if S is an element of B. If S is not an element of B, but would have `P` as a parent if it were, searchTree returns: `(FALSE, P, Pointer(P→S))`. `Pointer(P→S)` specifies whether S is a left or right child of `P`. This function relies upon the specification of the function g.
```
      IF( first entry of M is TRUE)
         Increment S.counter
```
This step simply increases the value of the counter corresponding to the string, S.
```
      ELSE
         insertNode(S, M, B)
```
insertNode creates a new node for S, with `P` for a parent. If B is empty, `insertNodes` creates the root of a binary tree. I intend this function to simply use the data generated by the `searchTree` function, so this step does not involve another binary search to choose a location for S.
```
      Increment line
```

Once the whole file has been read, and a very large binary search tree has been constructed, the program should print each token's name and the value of the counter. I suppose this last should be represented as an array of two-tuples.

[1] An implementation is available on Wikipedia